LICM Assignment

Amanjot Singh matr. 152792
Leonardo Temperanza matr. 152831

Di seguito è riportata la rappresentazione intermedia della funzione foo contenuta nel file test/Loop.ll:

```
define void @foo(i32 %0, i32 %1) {
  br label %3

3:                                                ; preds = %15, %2
  %.05 = phi i32 [ 0, %2 ], [ %19, %15 ]
  %.04 = phi i32 [ 0, %2 ], [ %17, %15 ]
  %.03 = phi i32 [ 0, %2 ], [ %16, %15 ]
  %.01 = phi i32 [ 9, %2 ], [ %.1, %15 ]
  %.0 = phi i32 [ %1, %2 ], [ %4, %15 ]
  %4 = add nsw i32 %.0, 1
  %5 = add nsw i32 %0, 3                          ; LI ; hoist
  %6 = add nsw i32 %0, 7                          ; LI ; hoist
  %7 = icmp slt i32 %4, 5
  br i1 %7, label %8, label %11

8:                                                ; preds = %3
  %9 = add nsw i32 %.01, 2
  %10 = add nsw i32 %0, 3                         ; LI ; hoist
  br label %15

11:                                               ; preds = %3
  %12 = sub nsw i32 %.01, 1
  %13 = add nsw i32 %0, 4                         ; LI ; hoist
  %14 = icmp sge i32 %4, 10
  br i1 %14, label %20, label %15

15:                                               ; preds = %11, %8
  %.02 = phi i32 [ %10, %8 ], [ %13, %11 ]
  %.1 = phi i32 [ %9, %8 ], [ %12, %11 ]
  %16 = add nsw i32 %5, 7                         ; LI ; hoist
  %17 = add nsw i32 %.02, 2
  %18 = add nsw i32 %0, 7                         ; LI ; hoist
  %19 = add nsw i32 %6, 5                         ; LI ; hoist
  br label %3

20:                                               ; preds = %11
  %21 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([25 x i8], [25 x i8]* @.str,
i64 0, i64 0), i32 %12, i32 %13, i32 %.03, i32 %.04, i32 %6, i32 %.05, i32 %5, i32 %4)
  ret void
}
```

Si può notare, innanzitutto, che i blocchi che dominano tutte le uscite del loop sono: %3 e %11; dunque le istruzioni contenute all'interno di tutti gli altri blocchi non possono essere spostate (a meno che non definiscano variabili dead, ovvero che non vengono usate al di fuori dal loop). Di fianco a ciascuna istruzione loop-invariant è presente un commento "LI".

Per definizione, un'istruzione loop-invariant è considerata tale se:

1. Tutte le definizioni che raggiungono l'istruzione si trovano fuori dal loop (o sono costanti)

2. C'è esattamente una reaching definition, e si tratta di un'istruzione loop-invariant

LLVM utilizza la forma SSA per la propria IR, perciò vi sono alcune garanzie implicite grazie a tale rappresentazione. Una di queste è il fatto che esiste sempre un'unica reaching definition rispetto ad un determinato uso, quindi effettivamente le condizioni sono due (per ciascun operando):

1. La sua reaching definition si trova fuori dal loop, oppure tale operando è costante
2. La sua reaching definition è un'istruzione che è altrettanto loop-invariant

Considerando che %0 non è utilizzato al di fuori del loop, le istruzioni che fanno uso di %0 possono essere spostate.

I registri SSA utilizzati al di fuori del loop sono: %12, %13, %.03, %.04, %6, %.05; tutti gli altri sono dead. Tentendo a mente la considerazione precedente relativa alla dominanza, solamente le istruzioni marcate con il commento "; hoist" sono spostabili.

Implementazione del passo

Il passo di trasformazione consiste in due passate distinte:

1. Raccolta di informazioni sulle istruzioni del loop; in particolare quali sono spostabili
2. Spostamento vero e proprio delle istruzioni al pre-header del loop

Il motivo per il quale devono necessariamente essere due pass separate è per fare in modo che si possa operare sulle istruzioni durante la prima fase dell'algoritmo senza dover gestire spostamenti. Una volta che le istruzioni sono annotate, si procede con la seconda fase dove si svolge una Depth First Search dei blocchi del loop a partire dal blocco header. Una Depth First Search permette di spostare un'istruzione prima dei suoi usi.

La prima fase invece, in particolare per quanto riguarda la loop-invariance, lascia spazio ad alternative di implementazione. Sono stati provati 3 approcci:

1. Funzione ricorsiva: ciascuna istruzione chiama la funzione ricorsivamente per i propri operandi, se questi sono loop-invariant allora anche l'istruzione di partenza lo è. I casi base della ricorsione sono istruzioni con side-effect (chiamate a funzione, store), costanti e argomenti di una procedura.
2. Hash-table: la chiave è un puntatore a llvm::Instruction e il valore associato è un bool, oppure inesistente. Questo permette teoricamente di risparmiare calcoli ripetuti, ma a differenza dell'implementazione ricorsiva richiede una visita Depth First Search del Control Flow Graph, in modo tale da incontrare le definizioni prima degli usi.

3. Metadati LLVM: come l'approccio hash-table, ma risparmiando in spazio e/o tempo (dipendentemente dall'implementazione della hash-table). Anche questo caso necessita una visita DFS.

È stato misurato il tempo occupato dai 3 approcci utilizzando un benchmark di dimensioni medio/grandi, ottenuto compilando il compilatore Odin (https://github.com/odin-lang) con Clang, usando le opzioni -emit-llvm e -O0. Si tratta di un progetto da circa 80.000 righe di codice che utilizza una sola unità di compilazione, e il codice intermedio prodotto è di circa 80MB. Tale codice è stato poi ottimizzato con le pass -mem2reg e -loop-simplify. Le misurazioni, effettuate con l'opzione -time-passes del comando opt, hanno portato ai seguenti risultati su un i5-9600KF (una media su 10 misurazioni). Si ricorda che il pass è stato compilato in -O2.

Wall Time	Nome
0.05084s	LICM (Ricorsiva)
0.05310s	LICM (Hash-Table), stb_ds
0.05665s	LICM (Metadati)